Cyclone Game Engine

Submitted by David Beirne 2006

Supervised by Dr. Zhijie Xu

Disclaimer

This report is submitted as part requirement for the BSc Degree in Computer Games Programming at the University of Huddersfield. It is substantially the result of my own work except where explicitly indicated in the text. This report may be freely copied and distributed provided the source is explicitly acknowledged.

Contents

EXECUTIVE SUMMARY **6**

1. INTRODUCTION **8**

2. BACKGROUND **8**

2.1 Multiplatform game engines *8*

2.2 API Abstraction *9*

2.3 Shaders *10*

3. ANALYSIS & DESIGN **15**

3.1 HARDWARE ANALYSIS 15

3.1.1 Xbox *16*

3.1.2 Target PC *16*

3.2 ANALYSIS OF EXISTING GAME ENGINES 17

3.2.1 OGRE *17*

3.2.2 Cry Engine *18*

4. IMPLEMENTATION **19**

4.1 ENGINE STRUCTURE 19

4.2 BASE 19

4.2 APPLICATION 20

4.3 ENTITY 21

4.3.1 Vectors *21*

4.3.2 Matrices *22*

World Transform *25*

4.4 CAMERA 25

View Transform *26*

Projection Transform *27*

4.4.1 FRUSTUM 30

4.5 SCENE 31

4.5.1 Scene Node *31*

4.5.2 Scene Graph *32*

4.5.3 Adding & Removing Scene Nodes *33*

4.5.5 Saving & Loading *34*

4.6 GRAPHICS 36

4.6.1 2D Graphics *36*

4.6.2 3D Graphics *37*

4.7 PHYSICS 40
4.7.1 Physics Manager *41*
4.8 SOUND 43
4.9 INPUT 43
4.9.1 Xbox Input *43*
4.9.2 PC Input *45*
4.9.3 Common Input Interface *46*
5.0 RESOURCE MANAGERS 46
5.1. Sprite Manager *47*
5.2. Sound Manager *47*
5.3. Physics Manager *48*
5.4. Animation Manager *48*
5.5. Mesh Manager *48*
5.6. Texture Manager *49*
5.6.1 Textures *49*
5.7 Cube Map Manager *49*
5.7.1 Cube Maps *49*
5.8 Volume Texture Manager *50*
5.8.1 Volume Textures *50*
5.9. Effect Manager *50*
5.9.1 Effects *50*
6.0. Material Manager *53*
6.0.1 Materials *53*
6.1. Subset Manager *54*
6.1.1 Subsets *54*
6.1.2 Rendering *55*

7. TESTING 56

7.1 TOWN OF THE DEAD 56
7.2 VIEWTOPIA 56

8. CONCLUSIONS, EVALUATION & FURTHER WORK 57

9. REFERENCES 58

Referenced Images: *59*

10. BIBLIOGRAPHY 60

CROSS-PLATFORM GAME PROGRAMMING 60
GAME PROGRAMMING GEMS 60

GAME PROGRAMMING GEMS 2 60
GAME PROGRAMMING GEMS 3 60
GAME PROGRAMMING GEMS 4 60
GAME PROGRAMMING GEMS 5 60
PROGRAMMING VERTEX AND PIXEL SHADERS 61
SHADERX^3 ADVANCED RENDERING WITH DIRECTX AND OPENGL 61
MATHEMATICS FOR 3D GAME PROGRAMMING & COMPUTER GRAPHICS 61
REAL-TIME 3D TERRAIN ENGINES USING C++ AND DIRECTX 9 61
3D GAME ENGINE DESIGN 61
GAME PHYSICS 61

11. APPENDICES 62

APPENDIX 1: XBOX HARDWARE SPECIFICATION 62
APPENDIX 2: UML CLASS DIAGRAM 62
APPENDIX 3: TOWN OF THE DEAD USER MANUAL 64
Controlling the player: 64
APPENDIX 4 : VIEWTOPIA USER MANUAL 65
Using the Menus: 65
Controlling the camera: 65
APPENDIX 5: CODE LISTING 66

Executive Summary

Cyclone is a powerful and robust game development tool. Incorporating advanced rendering physics and cross platform game development into a single bridging multi purpose and highly dynamic, programmer-friendly library. Utilizing cutting edge techniques and technologies to provide a highly expansive and efficient game generation process, Cyclone delivers the high performance and highly structured backbone for any game developing project.

As an Engine, Cyclones original structure simultaneously produces code for multiple platforms. Each function not only works for PC titles, but also for the Microsoft Xbox games console. This can effectively cut the development time in half or double a games platform scope. Commercially this would open both the huge PC Gaming Market, and also the larger Console Gaming Market to a single product. No time is required to port a game from one platform to the other, and the need for expensive console development kits is greatly reduced.

The Cyclone Engine also integrates the Tokamk Physics Library. A proven and successful game physics engine. This enables pseudo real world forces to act upon the in-games objects, increasing realism and game-play emersion. These Physics can be used in a variety of different ways, from rolling and bouncing a ball to stacking and knocking down crates. The Physics empower the users to create high quality and modern gaming experiences, replacing repetitive and static animations with real world reactions. These Physics techniques can interact with each other to form complex and structured mechanisms; adding chaos like reality and life like feedback.

Cyclone also incorporates a powerful material manager and editor. Not only does it give the freedom for adding pre-constructed models, but programmers can then take these much further. Using advanced rendering techniques and optimisation, a model is broken into visible and invisible subsets. This enables the engine to

only process the visible materials. The material editor allows objects textures to be changed within the code or even in real time! Materials can also utilise programmable vertex and pixel shaders in the form of FX files. This means cutting edge visual effects utilising the programmable GPU to its full potential and delivering the high quality graphics of most commercial engines. The material editor means Cyclone offers a unique bridge between professional modelling tools, such as 3D Studio Max or Maya and advanced Shader tools such as FX Composer or Render Monkey.

Cyclone is the sum of many interactive and innovative components – too many, in fact, to cover in this short summary please take a look at the more complete documentation that follows and the interactive demonstrations and games available on the enclosed CD.

1. Introduction

Cyclone is a cross-platform Game Engine for Microsoft DirectX 9 compatible PCs and the Microsoft Xbox games console. A Game Engine is the reusable core software component library used to create one or more videogames. Cyclone handles graphical rendering facilities (2D and 3D), Input from devices including Keyboard, mouse, joystick and game pads (such as those for the Xbox), Collision detection, Physics, 3d sound, and programmable shader support. The purpose of Cyclone is to provide a platform abstraction so that development for Xbox and PC is possible with no substantial change to game source code – allowing a single game to be compiled and run across the two platforms.

“With nearly every game on the market today being released simultaneously on all platforms, the need for a good cross-platform development strategy is crucially important. Every hour spent reinventing the wheel is an hour wasted. Or to put it another way, every hour spent writing cross-platform code constitutes three saved hours.” (Steven Goodwin (2005), Cross-Platform Game Programming)

2. Background

2.1 Multiplatform game engines

While re-usable sections of game code, such as sprite loading and rendering libraries were used extensively throughout the games industry in its early years, “The term ‘game engine’ only came about in the mid 1990’s” (Wikkipedia (2001), Game Engine Definition). The phrase coined by John Carmack, lead programmer at the then highly influential Id Software which created and popularized the First Person Shooter genre. “Such was the popularity of id Software's *Doom* and *Quake* games that rather than work from scratch, other developers licensed the core portions of the software and designed their own graphics, characters, weapons and levels—the ‘game content’ or ‘game assets’.

Later games, such as *Quake III Arena* and Epic Games' 1998 *Unreal* were designed with this approach in mind, with the engine and content developed separately. The licensing of such technology has proved to be a useful auxiliary revenue stream for some game developers, as a single license for a high-end commercial game engine can range from US $10,000 to $3,750,000(War Craft 3) and the number of licensees reaching several tens of companies (for the Unreal engine). At the very least, reusable engines make developing game sequels much easier and faster, a valuable advantage in the competitive computer game industry." (Wikkipedia (2003), Game Engine Definition).

"If the game engine is like our analogous car engine, think of the source code as not just the body of the car but some of the other parts as well -- maybe the transmission, the wheels, and anything else not inherently part of the engine. If you're repainting the car, the car looks different but has the same kind of power behind it. You might go a little further and put a high-end transmission on it. All of this would be much like modifying the game: the engine is still there and untouched, but you've changed what it's powering a little bit so that it does different things or looks a little different."
(3dPlanet (2001), What's a Game Engine?).

2.2 API Abstraction

"Abstraction means to generalise, ignoring details and specifics. For programmers, this means being able to write code without being concerned about the hardware, and knowing in advance that the behaviour of every platform will be identical – not similar but identical." (Steven Goodwin (2005), Cross-Platform Game Programming)

In order to provide a stable Cross-Platform Application Programmer Interface (API), Cyclone provides a single interface to both the Xbox Development Kit (XDK) and DirectX 9. The following parts of each API must therefore be considered:

“**Game**:
The Game should be programmed in a generic manner, using only the abstraction provided by the engine.

Engine:
Although parts of the engine will be platform specific (for example, the graphics driver), the majority of the code (such as geometry calculations) should be common. The interface that the engine presents to the rest of the world must therefore be common too.” (Steven Goodwin (2005), Cross-Platform Game Programming)

This involves glossing over any differences between the DirectX 9 SDK and the XDK and presenting a common and stable black-boxed interface to the rest of the world creating a fully abstracted API.

2.3 Shaders

Before the release of DirectX 8.0 compatible hardware in early 2002, the functionality of the graphics pipeline was fixed into the graphics card hardware. With advancing technology and the popularity of the Render-Man software package developed a subset of George Lucas’ Industrial Light and Magic (ILM) for the films “Terminator 2” and “the Abyss” known as Pixar. ILM continued to utilise RenderMan in its “Jurassic Park” and “Star Wars” films while Pixar went on to create the first popular full-length computer generated animated movies “Toy Story”, “A Bugs Life” and “Finding Nemo”. (Pixar (2006), Artists Tools – What is RenderMan?)

Such was the popularity of these films that graphics card companies such as ATI and Nvidea began to create graphics processing units (GPU’s) with a programmable interface while software companies such as Microsoft built programmable shaders into the DirectX API pipeline. Early shaders could only be

written in assembly language but with the almost simultaneous releases of Microsofts High Level Shader Language (HLSL) and Nvidia's C for Graphics (Cg) in 2002, came the ability for programmers to develop shaders in a C-like language.

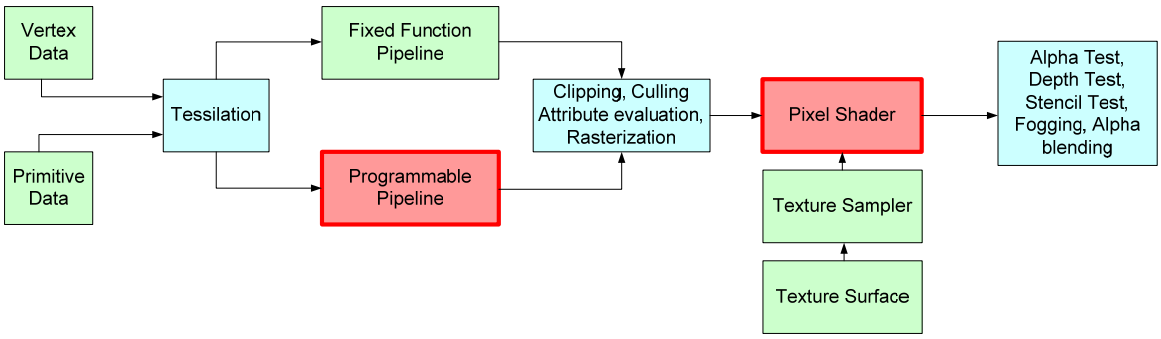


Figure 1 : Direct3D pipeline

This figure shows how the programmable vertex and pixel shaders were added to the DirectX rendering pipeline where the red boxes indicate the additional units.

2.3.1 Vertex Shaders

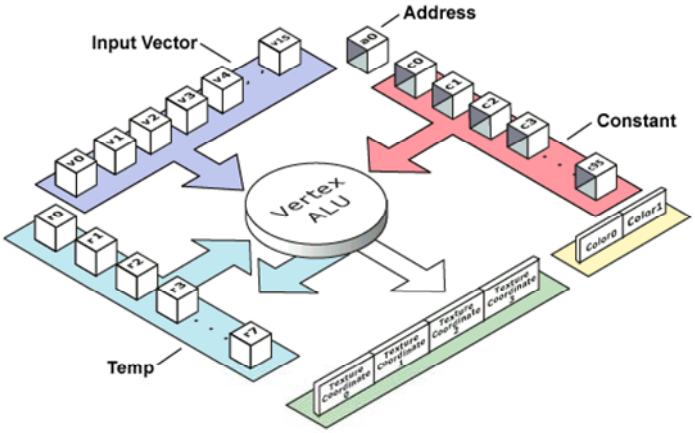


Figure 2 : GPU Vertex Shader Unit

(Image from http://www.zfx.info/Tutorials.php?ID=71)

A programmable vertex shader replaces the transformation and lighting module in the Microsoft Direct3D geometry pipeline (see above). In effect, state information regarding transformation and lighting operations are ignored and replaced with the instructions of the shader running on the Vertex shader Unit.

A vertex shader is a small program which runs on a per-vertex basis; it must generate vertex positions in homogeneous clip space, but can also be used to generate texture coordinates, vertex colours, normal data, fog factors etc.

The standard graphics pipeline processes the vertices output by the shader, including the following tasks.

- Primitive assembly
- Clipping against the frustum and user clipping planes
- Homogeneous divide
- Viewport scaling
- Back-face and viewport culling
- Triangle setup
- Rasterization

Programmable geometry is a mode within the Direct3D application programming interface (API). When it the vertex shader is enabled, it partially replaces the vertex pipeline. When it is disabled, the API switches back to fixed function vertex processing. Execution of vertex shaders does not change the internal Direct3D state, and no Direct3D state is available for shaders.

2.3.2 Pixel Shaders

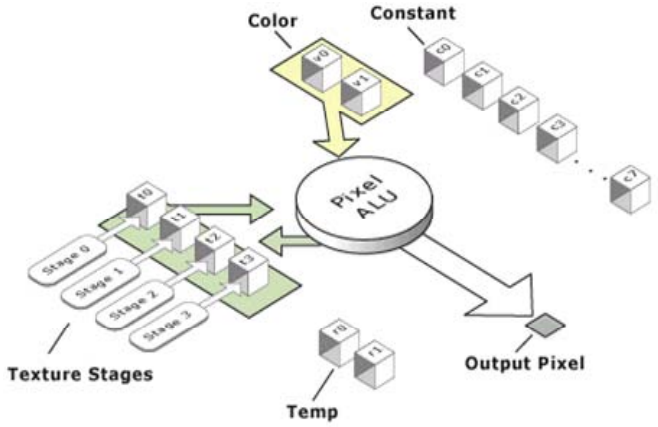


Figure 3 : GPU Pixel Shader Unit

(Image from http://www.zfx.info/Tutorials.php?ID=71)

Pixel processing is performed by pixel shaders on individual pixels. Pixel shaders can work by themselves or in concert with vertex shaders and streams.

The sequence of operations in the pixel pipeline is as follows:

- Triangle setup
- Pixel shader (replaces fixed format multitexture)
- Iterate colours, texcoords, etc.
- Sample textures
- Texture/colour blend
- Fog blend
- Alpha, stencil, depth testing
- Frame-buffer blend

The inputs to the pixel shader come from the outputs of the vertex shader. Registers v0 and v1 contain the vertex colours from vertex shader output registers oD0 and oD1. The textures in the colour stages, referenced by pixel shader instructions like tex t0, are sampled using the texture coordinates in the vertex shader output register corresponding to that stage, like oT0. Pixel shaders

use colour and alpha-blending instructions and texture-addressing instructions to manipulate these inputs and calculate a result. The result emitted by the pixel shader is the contents of register r0 or the output pixel colour. Whatever it contains when the shader completes processing is sent to the fog stage and render target blender for further processing. Vertex shader outputs can provide pixel shader inputs.

2.3.3 Effect Files

Cyclone needs to be able to use many different visual effects for various objects and environmental conditions, and the need to support them across a variety of hardware that often require different implementations.

DirectX 8 introduced the concept of Effect Files, a file format allowing for the description of rendering techniques outside of the compiled application, which can be loaded at run-time and applied to objects during rendering. Separating the rendering code in this way has a number of significant benefits:

- Rendering techniques are decoupled from the engine, meaning less hard coded rendering code and greater flexibility, as new visual effects can be added without modifying application code.

- An effect file can contain several implementations of a given technique to allow for varying hardware.

- By separating the programming of visual effects from the application code, effect files can potentially become part of the art pipeline, accompanying incoming geometry and associated media with the specifics of how they are to be rendered.

Effect files are made up of three basic components:

2.3.3.1 Parameter declarations:

These are named values that may be set by the application prior to rendering, to be used by the functions and techniques defined in the effect file. These are how the application will communicate values necessary for rendering, such as transformation matrices and lighting parameters.

2.3.3.2 Techniques:

A technique is a definition of how to render an object. It includes all rendering and texture state settings that need to be set prior to rendering, as well as vertex and pixel Shader declarations. A technique may consist of one or more rendering passes to support multi-pass rendering, each pass containing its own state and Shader definitions.

2.3.3.3 Functions:

Functions are written in High Level Shader Language (HLSL), a C-style language which can be used as an alternative to writing Shaders in assembly language. Functions can be compiled into vertex or pixel Shaders, may be called from other functions, or used in many other ways.
More information on effect files can be found later sections.

3. Analysis & Design

3.1 Hardware analysis

One of the key features of this project is that it is required to run on the Microsoft Xbox Games console as well as a PC. Being a games console, the Xbox has a fixed hardware specification:

3.1.1 Xbox

CPU: 32-bit 733 MHz Intel Pentium III
GPU: 233 MHz Custom NVIDIA NV2A
Memory: 64 Mb DDR SDRAM 200 MHz
Storage: 2-5x DVD, 8 GB Hard Disc
Network: 10/100 base-T Ethernet
API: XDK (DirectX 8.1 based).*

*(*See Appendix 1 for more detailed hardware specification)*

3.1.2 Target PC

CPU: 32-bit 1.30 GHz AMD XP 3000 +
GPU: 250 MHz Radeon 9200 +
Memory: 512 MB DDR SDRAM 200 MHz +
Storage: 1-12x CD, 4 GB HDD +
Network: 10/100base-T Ethernet
API: DirectX 9.0 SDK

The advantage of working to a fixed hardware specification is that, as with all games consoles, there is a large user-base which ranges from the technology savvy “hardcore gamers” who are likely to understand and appreciate the hardware specification of both their PC and console to the more casual gamer who is unlikely to understand any technological issues. A console allows this broad range of users to have access to software which will immediately run on the hardware without any need for installation or any prior knowledge of computing.

The disadvantage of working to the Xbox as the benchmark hardware is that the Xbox is now an aging platform. While a few years ago it represented the cutting-edge of gaming hardware, comparable to the top level of gaming PC available, today it represents more of a lower-to-mid range PC with its fixed hardware

which cannot be upgraded. For this reason, the PC version of Cyclone may be constrained by the Xbox's capabilities.

3.2 Analysis of Existing Game Engines

3.2.1 OGRE

Ogre (**O**bject-Oriented **G**raphics **R**endering **E**ngine) is a popular open source Engine which has been in development for both the DirectX and OpenGL API's by Tours Knot Software since 2002. Being open source and very comprehensibly documented, the OGRE engine can be studied and distributed freely under the GNU Lesser General Public Licence (LGPL).

3.2.1.1 Lessons from the OGRE engine.

OGRE has been studied extensively during later half of the production of Cyclone and many parallels can be drawn between the engines. Some deliberate and some coincidental. The main similarity between the two engines is the highly object oriented and design-driven approach they both take. OGRE also inspired the implementation of the Resource Managers found in Cyclone although the implementation is very different.

Cyclone and OGRE also share a similar approach to material management. OGRE creates its own material and model formats, with material attributes and effects described within a single text document, with effects in the form of either HLSL, Assembly, Cg or GLSL which are then applied to a whole mesh model within the code of the game. Cyclone has taken a similar approach with its Material Library file. However Cyclone only supports HLSL and Assembly shaders stored in FX files – this is because it does not have to run on anything other then the DirectX API where the use of Cg is discouraged and GLSL is not possible. Cyclone also offers some material features beyond the scope of OGRE. While OGRE only allows an effect or material to be applied to a mesh as a whole through the games code, Cyclone not only allows this but also allows an

effect or material to be applied to a subset of a mesh. Cyclone can also apply materials at runtime as demonstrated in its presentation game.

3.2.2 Cry Engine

They Cry Engine by CryTek is of particular significance to this research because it is a relatively recent PC game engine which has just been ported from PC to Xbox for the releases of “Far Cry: instincts”. While its PC counterpart is API independent (i.e. runs on both OpenGL and DirectX), the leaner Xbox version is DirectX only. Another interesting feature of the Xbox game is that it has a level editing tool, allowing the player to interact more directly with the engine then in a gameplay situation.

3.2.1.2 Lessons from the Cry Engine.

Having used the Cry Engine level editor on both the PC and Xbox versions of the game, as well as playing the games themselves and reading any available documentation on the engine, there are some useful lessons that can be learned from it. Firstly the Cry Engine is fully commercial and therefore protected by copyright laws so many of the internal workings of the engine remain hidden from the user. However, the creators of the Cry Engine did give a talk at the 2005 Game Developers Conference (GDC) on how their engine uses DirectX to maximise performance.

The entire document is available for download on ATI’s website (http://www.ati.com/developer/gdc/D3DTutorial08_FarCryAndDX9.pdf) but the main point utilised by the Cyclone engine is the geometry instancing and batching described on pages 9-16 where all repeated models are instances of a single mesh with unique position scale and rotation.

Another interesting feature of the CryEngine which is not available in the OGRE engine is the collision detection and Physics experienced within the game. FarCry along with many other recently releases games (such as Half-Life 2) use

a third-party physics middleware solution such as Havok which allows in-game objects to react within a real-time physics simulator. This is a very interesting feature which was adopted by Cyclone using the freely distributable Tokamak physics SDK (see Physics section).

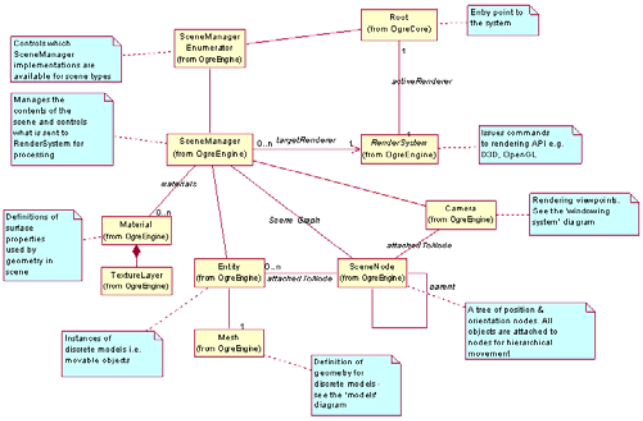
4. Implementation

4.1 Engine Structure

Cyclone has been designed to follow the Object Oriented structure as closely as possible. The advantage of the Object Oriented approach is not only the reusability and modularity of code but it also breaks the large system down into understandable components. One of the main aims of a game engine is to be comprehensible to multiple users as it could form the backbone of the development process for a team of programmers of varying ability. What follows is a brief overview of the main components of the engine. For a more in-depth look at the implementation see the attached UML (Appendix 2) and source listings on the included CD (or visit http://www.acidgames.net/Cyclone/Source).

4.2 Base

A common feature of many game engines is a Base (or Root) class:



A schematic UML overview of the core objects of OGRE

(Image from http://www.ogre3d.org/docs/manual/images/uml-overview.png)

“The 'Root' object is the entry point to the OGRE system. This object MUST be the first one to be created, and the last one to be destroyed.” (*Steve Streeting (2005)* OGRE Manual).

While in the OGRE engine the Root is decoupled from a specific API, the Cyclone a Base class has been defined to encapsulate the core elements of the DirectX API which must be shared by other classes within the engine, giving access to shared elements such as the graphics, sound, and input devices. The Base also includes the code required to run, maintain and close the API device objects such as the Direct3D Device.

Along with the device objects, the base contains a series of Managers. The managers will be described in more detail in a later section, but the key concept behind each manager is to ensure that any external resources are maintained in an organised and normalised fashion.

As well as the managers and device access objects, the Base also includes useful methods for debugging, such as those to log information or errors to an html file and the ability to compute the memory footprint of the game, file managing functions, and the ability to display platform independent message box interrupts.

The main reason for having a Base class is so that the engine is not dependent on global structures

4.2 Application

The Application entry point is where any C++ program begins. Under windows, the application entry point is known as the WinMain() and main() on Xbox. The

application entry point represents a single function where the program begins and ends.

On the Xbox, only one program can be running at any one time; however Windows is a multiprogramming operating system - this means that many programs can be running at any one time all taking turns to use the same processor. In order to fane parallelism, Windows has incorporated a messaging system between windows therefore the Application must handle the Windows messaging procedure. Windows also allows a standard window to be position and scaled at any time, the application class handles this.

4.3 Entity

In the Cyclone Game Engine, an Entity represents a unique object within 3d space. The Entity encapsulates the translation, rotation and scaling of objects within 3d space along with the matrices on which the transformation operations are carried out. Each entity also contains a pointer to the single instance of the Base class which provided it and any derived class with access to the functions and references within the base.

4.3.1 Vectors

Each Entity instance contains three 3D vectors. A 3D vector (in its simplest form) is a set of 3 floating point numbers which can represent a 3D coordinate comprising of an X, Y and Z component. The Entity has a 3D Vector to store its Position (i.e. the Cartesian co-ordinate of the entity in 3d space), its Rotation (the amount by which the entity is rotated about the X, Y and Z axis) and its Scale (the factor by which the entity is scaled on the X, Y and Z axis). While it is simple enough to define a custom vector structure, the Entity uses the DirectX native D3DXVECTOR3 structure because DirectX provides many useful methods which require or return a D3DXVECTOR3. The D3DXVECTOR3 may also take advantage of the useful operator overloads and type-casting extensions defined in the d3dx9math.inl / d3dx8math.inl library.

```
typedef struct D3DXVECTOR3 {
    FLOAT x;
    FLOAT y;
    FLOAT z;
} D3DXVECTOR3, *LPD3DXVECTOR3;
```

4.3.2 Matrices

Each Entity instance contains six 4x4 matrices. A matrix represents an array of sixteen floating point numbers and is used to calculate the position, scale and rotation of the entity in world space.

```
typedef struct D3DXMATRIX {
    FLOAT _11, FLOAT _12, FLOAT _13, FLOAT _14,
        FLOAT _21, FLOAT _22, FLOAT _23, FLOAT _24,
        FLOAT _31, FLOAT _32, FLOAT _33, FLOAT _34,
        FLOAT _41, FLOAT _42, FLOAT _43, FLOAT _44 );
} D3DXMATRIX;
```

4.3.2.1 Matrix Transforms

Matrix transformations can be used to express the location of an object relative to another object, rotate and scale objects in 3d Space and to change viewing positions, directions, and perspectives.

Translation

Any 3D point (x, y, z) can be translated into another point (x', y', z') using a 4x4 matrix. The following transform translates the point (x, y, z) to a new point (x', y',z').

$$[x'y'z'1] = [x\ y\ z\ 1]\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ T_x & T_y & T_z & 1 \end{bmatrix}$$

As with the Vector, Cyclone uses the DirectX standard D3DXMATRIX type because the D3DX utility library supplies the D3DXMatrixTranslation function.

Scaling

The following transform scales the point (x, y, z) by arbitrary values in the x-, y-, and z-directions to a new point (x', y', z').

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotation

Cyclone uses the DirectX standard left-handed Axis. The following matrix rotates the point (x, y, z) around the x-axis, producing a new point (x', y', z').

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\theta & \sin\theta & 0 \\ 0 & -\sin\theta & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following matrix rotates the point around the y-axis.

$$\begin{bmatrix} x' & y' & z' & 1 \end{bmatrix} = \begin{bmatrix} x & y & z & 1 \end{bmatrix} \begin{bmatrix} \cos\theta & 0 & -\sin\theta & 0 \\ 0 & 1 & 0 & 0 \\ \sin\theta & 0 & \cos\theta & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

The following matrix rotates the point around the z-axis.

$$[x'y'z'1] = [x\ y\ z\ 1]\begin{bmatrix} \cos\theta & \sin\theta & 0 & 0 \\ -\sin\theta & \cos\theta & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Here theta represents the angle of rotation, in radians. Angles are measured clockwise when looking along the rotation axis toward the origin.

Cyclone utilizes the D3DXMatrixRotationYawPitchRoll() function supplied by the D3DX utility library to create a single rotation matrix for each Entity.

Concatenating Matrices

One advantage of using matrices is that the effects of two or more matrices can be combined by multiplying them. This means that, to rotate a model and then translate it to some location, there is no need to apply two matrices. Instead the rotation and translation matrices are multiplied together to produce a composite matrix that contains all their effects. This process, called matrix concatenation, can be written with the following formula.

$$C = M_1 \cdot M_2 \cdot M_{n-1} \cdot M_n$$

In this formula, C is the composite matrix being created, and M_1 through M_n are the individual matrices. In most cases, only two or three matrices are concatenated, but there is no limit.

Cyclone utilizes the D3DXMatrixMultiply() function to perform matrix multiplication. The order in which the matrix multiplication is performed is highly important. The preceding formula reflects the left-to-right rule of matrix concatenation. That is, the visible effects of the matrices that you use to create a composite matrix occur in left-to-right order. A typical world matrix is shown in the

following example where an object is rotated to some angle on the Y axis and positioned to an arbitrary location.

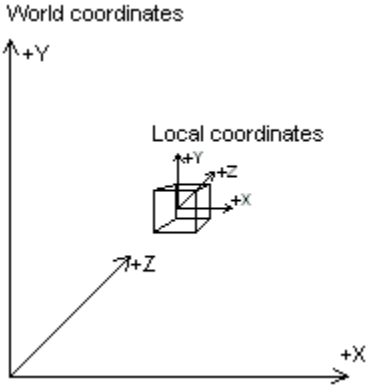
$$W = R_y \cdot T_w$$

In this formula, R_y is a matrix for rotation about the y-axis, and T_w is a translation to some position in world coordinates.

The order in which the matrices are multiplied is important because, unlike multiplying two scalar values, matrix multiplication is not commutative. Multiplying the matrices in the opposite order has the visual effect of translating the object to its world space position, and then rotating it around the world origin.

World Transform

A world transform changes coordinates from model space, where vertices are defined relative to a model's local origin, to World Space, where vertices are defined relative to an origin common to all the objects in a scene. Essentially the world transform places a model into the world. The following diagram illustrates the relationship between the world coordinate system and a model's local coordinate system:



4.4 Camera

A Camera in Cyclone represents an Entity through with the 3d scene can be viewed. For convenience, the engine creates a default camera when the Base is initialised (see Base section). The analogy of a camera is carried through in the

ability of the camera to be configured and manipulated in the same way as a physical film camera, with the ability to set the perspective, the near and far clipping distances (or range), and the ability to pan, dolly, roll, pitch and yaw.

It is possible to have multiple cameras within an environment. However it is encouraged that each camera be set as the Base camera before it is rendered so that any other elements of the scene have access to it if they require it – i.e. an Effect may need to access the projection matrix or view position of the current camera to render correctly or a NodeTree mesh (see NodeTree section) will need to access the current cameras view frustum (see frustum section). This design decision was made to keep the camera system open-ended so the programmer has the choice of how they want to use the camera.

View Transform

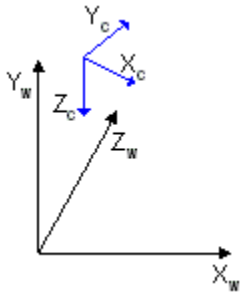
The view transform locates the viewer in world space, transforming vertices into camera space. In camera space, the camera, or viewer, is at the origin, looking in the positive z-direction. Direct3D (and subsequently Cyclone) uses a left-handed coordinate system, so z is positive into a scene. The view matrix relocates the objects in the world around a camera's position - the origin of camera space - and orientation.

There are many ways to create a view matrix. In all cases, the camera has some logical position and orientation in world space that is used as a starting point to create a view matrix that will be applied to the models in a scene. The view matrix translates and rotates objects to place them in camera space, where the camera is at the origin. One way to create a view matrix is to combine a translation matrix with rotation matrices for each axis. In this approach, the following general matrix formula applies.

$$V = T \cdot R_z \cdot R_y \cdot R_x$$

In this formula, V is the view matrix being created, T is a translation matrix that repositions objects in the world, and R_x through R_z are rotation matrices that

rotate objects along the x-, y-, and z-axis. The translation and rotation matrices are based on the camera's logical position and orientation in world space. So, if the camera's logical position in the world is <10,20,100>, the aim of the translation matrix is to move objects -10 units along the x-axis, -20 units along the y-axis, and -100 units along the z-axis. The rotation matrices in the formula are based on the camera's orientation, in terms of how much the axes of camera space are rotated out of alignment with world space. For example, if the camera is pointing straight down, its z-axis is 90 degrees (pi/2 radians) out of alignment with the z-axis of world space, as shown in the following illustration.



The rotation matrices apply rotations of equal, but opposite, magnitude to the models in the scene. The view matrix for this camera includes a rotation of -90 degrees around the x-axis. The rotation matrix is combined with the translation matrix to create a view matrix that adjusts the position and orientation of the objects in the scene so that their top is facing the camera, giving the appearance that the camera is above the model.
(msdn.microsoft.com)

Setting Up a View Matrix

Cyclone uses the D3DXMATRIXLookAtLH and D3DXMATRIXLookAtRH helper functions create a view matrix based on the camera location and a look-at point.

Projection Transform

The projection transformation can be thought of as choosing a lens for the camera. The projection matrix is typically a scale and perspective projection. The projection transformation converts the viewing frustum into a cuboid shape.

Because the near end of the viewing frustum is smaller than the far end, this has the effect of expanding objects that are near to the camera.

In the viewing frustum, the distance between the camera and the origin of the viewing transform space is defined arbitrarily as *D*, so the projection matrix looks like:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

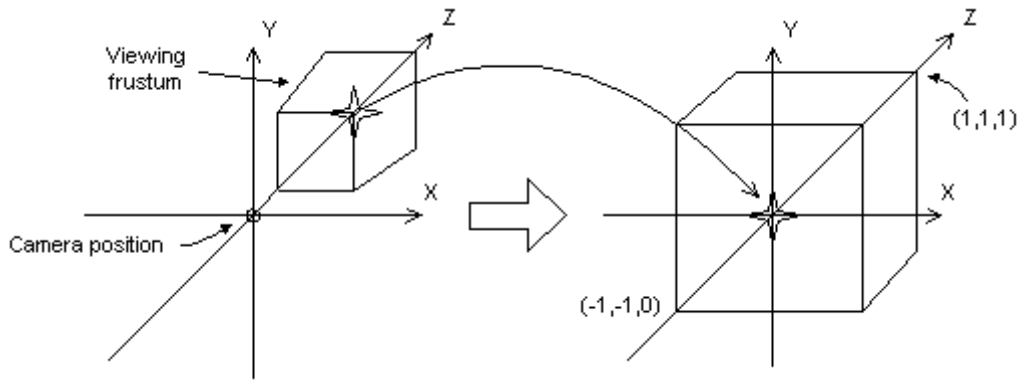
The viewing matrix translates the camera to the origin by translating in the z direction by - *D*. The translation matrix is as follows:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & -D & 1 \end{bmatrix}$$

Multiplying the translation matrix by the projection matrix (T*P) gives the composite projection matrix. It looks like:

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1/D \\ 0 & 0 & -D & 0 \end{bmatrix}$$

The perspective transform converts a viewing frustum into a new coordinate space. The frustum becomes cuboid and the origin moves from the upper-right corner of the scene to the centre.



In the perspective transform, the limits of the x- and y-directions are -1 and 1. The limits of the z-direction are 0 for the front plane and 1 for the back plane.

This matrix translates and scales objects based on a specified distance from the camera to the near clipping plane, but it does not consider the field of view (fov), and the z-values that it produces for objects in the distance can be nearly identical, making depth comparisons difficult. The following matrix addresses these issues, and it adjusts vertices to account for the aspect ratio of the viewport, making it a good choice for the perspective projection.

$$\begin{bmatrix} w & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & Q & 1 \\ 0 & 0 & -QZ_n & 0 \end{bmatrix}$$

In this matrix, Z_n is the z-value of the near clipping plane. The variables *w*, *h*, and *Q* have the following meanings. Note that fov_w and fov_k represent the viewport's horizontal and vertical fields of view, in radians.

$$w = \cot\left(fov_w/2\right)$$

$$h = \cot\left(fov_k/2\right)$$

$$Q = \frac{Z_f}{Z_f - Z_n}$$

Using field-of-view angles to define the x- and y-scaling coefficients is inconvenient so the viewport's horizontal and vertical dimensions (in camera space) are used instead. The following two formulas for *w* and *h* use the viewport's dimensions, and are equivalent to the preceding formulas.

$$w = \frac{2 \cdot Z_n}{V_w}$$

$$h = \frac{2 \cdot Z_n}{V_h}$$

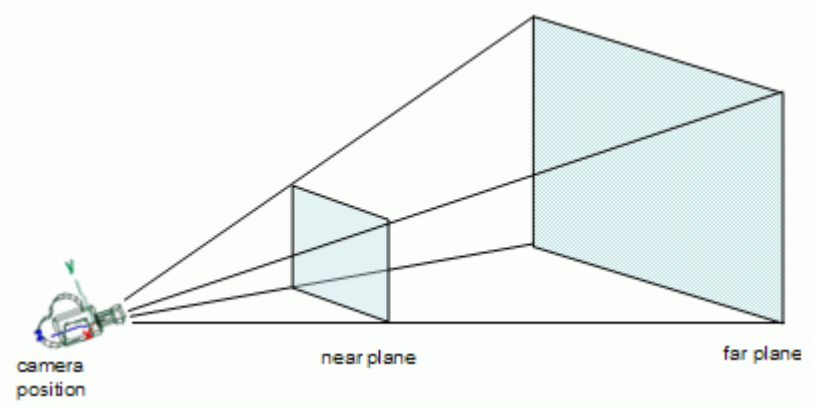
In these formulas, Z_n represents the position of the near clipping plane, and the V_w and V_h variables represent the width and height of the viewport, in camera space. As with the world and view transformations, you call the SetTransform method to set the projection transform.
(msdn.microsoft.com)

4.4.1 Frustum

Each camera contains a viewing frustum.

“In 3D computer graphics, the **viewing frustum** or **view frustum** is the region of space in the modeled world that may appear on the screen; it is the field of view of the notional camera. The exact shape of this region varies depending on what kind of camera lens is being simulated, but typically it is a frustum of a rectangular pyramid.” *(Wikkipedia (2004) Viewing Frustum Definition)*

The Frustum class constructs a rectangular pyramid from 6 planes representing the near, far, left, right, top and bottom clipping planes of the field of vision in front of the camera:



(Image from http://www.lighthouse3d.com/opengl/viewfrustum/images/vf.gif)

The frustum contains methods to check if a given point, sphere, cube or cuboid (rectangle to avoid confusion) is within the space of the frustum. This is calculated using the D3DXPlainDotCoord function to check the dot product of a point (or the points describing a shape) with the six sides of the frustum. If the point is inside, or any points of the shape are inside then the method returns true, otherwise it returns false. These methods are used by the Scene and NodeTree classes (see later sections) for culling. However these methods could also be used in other situations, such as AI, so the Frustum Class is kept separate from the camera.

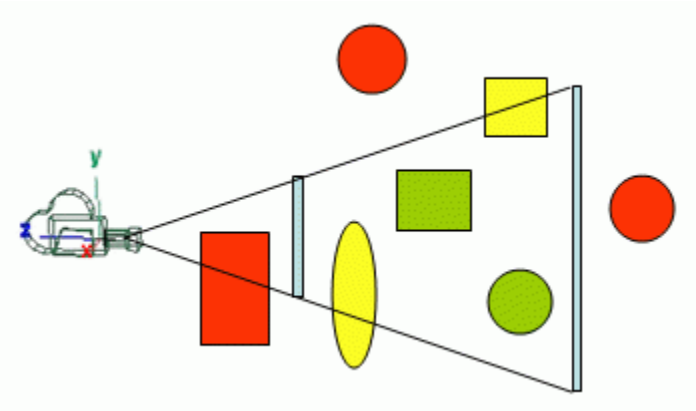
4.5 Scene

In Cyclone a Scene encapsulates a graph of Scene Nodes and the methods that may be applied on the scene graph, to load, save add modify and render a 3d scene consisting of Entities.

4.5.1 Scene Node

The Scene Node is a wrapper class for an Entity which also contains information on how the Scene Node (and essentially the entity pointed to within it) relates to the other items in the scene graph. The attributes that describe the relationship are pointers to other Scene Nodes which may be either the parent of the current node, its child or the next node in the (linked) list. Any node can be the parent or

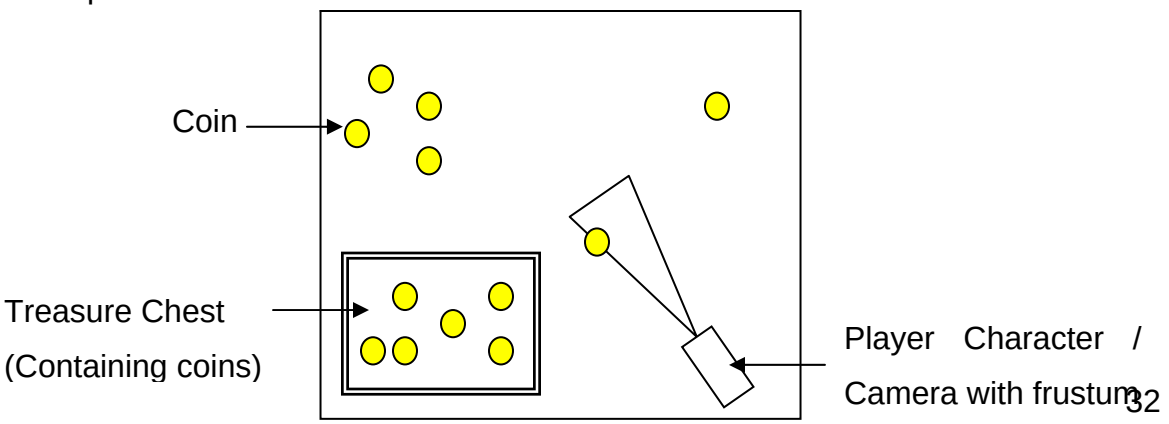
child of another provided that a cyclical reference is not generated (i.e. a Scene Node cannot be its own parent, child or cannot point to itself as the next node).



In the figure above the green objects represent entities totally inside the view frustum, the yellow objects are partially inside and would be rendered, whereas the red Items will not be rendered. Note that the green sphere is not visible (it is occluded by the yellow ellipse), but it will be rendered anyway because it is inside the view frustum.

4.5.2 Scene Graph

The reason for using the relationship information stored in each scene node is to reduce the amount of time taken for the Scene to decide which Scene Nodes to render. Whereas a simpler unmanaged scene would iteratively check every item in the 3d World against the cameras viewing frustum, the Scene managed by a Scene Graph would check the parent Node, if that is visible, check its children, and then go on to check the next Node the benefits of this can be best seen in an example:



In this example scene (viewed in plan) the player is standing in a room with 12 gold coins and a treasure chest. The player can only actually see 1 coin and the room he is standing in, due to the size of the room and his limited view frustum. To render this scene with no relationships between nodes would be the same as rendering a list:

Is Coin 1 visible?
No. So do not render coin 1.
Is Coin 2 visible?
No. So do not render coin 2.
Is coin 3 visible?
Yes, render coin 3.
Is Coin 4 Visible?
No.
…
Is the chest Visible?
No
Is Coin 5 visible?
No.
Etc...

This can become very time consuming and slow the games rendering speed to a crawl, particularly with large scenes. However in the diagram 6 of the 12 coins are within the chest and therefore could only be visible if the chest is visible. This is the perfect candidate for a parent, child relationship where each coin within the chest belongs to the chest. That way when the camera comes to check if the chest is visible and the answer is no, it automatically knows that the 6 coins in the chest are not visible either so does not have to test if they are within its frustum. Furthermore the room the player is standing in could be the child of a larger building, and the building a part of a city etc. The benefits of only checking if parent items are visible can be immense.

4.5.3 Adding & Removing Scene Nodes

While in the example above, the chest, coins and room were presumed to be already in the scene, the Scene class also has the methods to dynamically add to and remove items from the world at any time. This would be useful in many

gaming situations (such as a player picking up and dropping items) and could even be used in a world editing tool.

4.5.5 Saving & Loading

The scene also has the ability to Save and load a 3d world from a typical Excel spreadsheet or text file. In order to do this two files must be provided to describe the scene – a master item list and a map file. An optional third file may also be loaded containing a library of materials.

4.5.5.1 The Master Item List

The master Item list is a list of all things that could possibly be in the scene. This could vary from game to game, but a (very short) typical list may look like this:

Name	Model path	Texture Path	Category
Level	Demo Level\level.x		1
Zombie	Models\Zombie.x		0
Box	Models\ball.x		3
Ball	Models\ball.x		2
Light			5
Sound1	Sounds\Birds.wav		6

Things to notice here are that while each item in the list is unique some use the same model and others the same category. The Category filed tells the Scene what type of Entity to load if this Item is referenced in the map file for example 0 would be an Object, 1 a NodeTreeMesh, 3 a RigidBodyObject, 4 an AnimatedBodyObject, 5 a Light, 6 a Sound etc. The Scene will ignore empty fields.

5.5.5.2 The map file

While the master item list contains a list of all unique items that could possibly be in the Scene, the map contains a list of all of the Scene Nodes that are actually within the scene:

MIL Number	Position			Scale			Rotation			Bounding Type	Mass	Owner
	X	*Y*	*Z*	*X*	*Y*	*Z*	*X*	*Y*	*Z*			
0	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0	30	-1
1	0.0	60.0	0.0	1.0	1.0	1.0	0.0	20.0	0.0	0	30	-1
1	0.0	40.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	1	30	-1
2	-36.0	40.0	0.0	1.0	1.0	1.0	0.0	0.0	10.0	2	60	-1
1	36.0	40.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0	60	-1
2	0.0	400.0	-120.0	1.0	1.0	1.0	4.0	0.0	0.0	1	80	-1
3	0.0	400.0	120.0	1.0	1.0	1.0	0.0	0.0	0.0	0	80	-1
2	-40.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0	80	-1
1	-60.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	0	80	-1
4	0.0	0.0	0.0	1.0	1.0	1.0	0.0	0.0	0.0	1	80	-1

In the map, the MIL number refers to the Item in the Master Item List that corresponds to the current Scene Node, the position, scale and rotation orientate the node in the scene. The Bounding type can be 0, 1 or 2 which represent a bounding box sphere or cylinder which are calculated for any physics object (see physics section). The owner stores which items belong to others.

4.5.5.3 Material Library

The optional material library allows a series of materials defined externally of the game code; these can then be applied to subsets in real-time. The advantage of this is that a material defined in the material library may include an effect and or a vertex colour and or texture for a subset (see subset manger section) whereas the material defined within the models ".x" file may not contain effects.

Vertex Colour				Texture	Effect File
Red	Green	Blue	Alpha		
255	255	255	255	Materials\HDR\Logo.png	Materials\HDR\HDR.fx
10	255	10	50		
10	255	105	255	Materials\Skin.png	
255	255	255	255	Materials\Stone.png	Materials\Bump\BumpStone.fx

If a material defined in the material library only has a vertex colour, any subset with this material will be rendered with only flat vertex colours. If the material contains a texture, then the texture will be applied to the subset before it is

rendered, and a combination of the texture and material colour will be used. If the material has an effect, then the effect will either use the vertex colour and texture (if it is programmed to do so) or may override them. Cyclones SubsetManager (see later section) is programmed with a fallback so if the effect file does not work on the specified hardware, the texture and or material will be used.

4.6 Graphics

4.6.1 2D Graphics

4.6.1.1 Sprites

The term sprite refers to a 2D element rendered within a 3D environment. Sprites can be used for on screen displays, menus or even to make 2d games. In Cyclone, the sprite class constitutes an Entity with an ID3DXSprite instance and an array of frame structures.

```
//Frame structure
struct SpriteFrame
{
    TEXTURE image;
    float width;
    float height;
};
```

The frame represents an image loaded from any image format file into a TEXTURE which is a platform independent texture class defined in Cyclone.h and used for Textures throughout the engine. The width and height are also read from the image file and stored in the frame to be used in the collision detection and clicking methods. A Sprite can either be single-framed and static, or contain multiple frames. If a sprite has many frames, it will loop through them to play an animation. As Entities, Sprites can be positioned, scaled and rotated in screen space using the same matrices and vectors as the Entity class.

One other use of Sprites in Cyclone is as render targets. In the engine demonstration the normal 3d scene is rendered to the back buffer, as with a normal 3d application. However the material menus are rendered to a texture

which is then applied to the first frame of a sprite and rendered on top of the 3d scene. This allows the material menus to be rendered in real-time so any changes to material parameters can be seen immediately.

4.6.1.2 Text

Text output on the Xbox is incredibly limited, most commercial games do not use text in the final build but create sprites containing any text required. This is because writing Text to a surface using the standard DirectX methods is a surprisingly expensive operation so is discouraged. The standard text methods also vary between DirectX releases. For this reason, Cyclone supplies separate text classes for the two platforms. The platform dependent class is selected at compile-time by pre-processor code.

4.6.1.2.1 PC Text

The Text class for the pc loads a True-Type font (ttf), and creates a rectangle to write a character string to in screen-space.

4.6.1.2.2 Xbox Text

The Xbox encourages the use of its own bitmap font (.bmf) format for loading fonts, while this increases performance over true-type, it also adds the task of having to create and supply the bitmap font file. So as not to restrict programmers, the Cyclone Xbox font also provides the ability to load true-type fonts. The Xbox font writes a character string directly to the back-buffer which is then presented (at the end of each render cycle).

4.6.2 3D Graphics

4.6.2.1 Object

The Object Class encapsulates all methods and data attributed with loading, rendering and destroying 3d objects. The object must be stored in the DirectX native ".x" format. An object may be animated, using skin and bone or frame based animation or static. An object may contain instances of the included Mesh,

Animation, and AnimationSet classes which are composed of many smaller structures and classes (described below). The object also contains the methods involved with Rendering, animation, (none physics based) collision detection, and intersection involving picking and ray-casting.

4.6.2.1.1 Mesh

The Mesh class represents a Mesh-List or a number of Mesh-Lists loaded from a single DirectX (.x) mesh file. The Mesh class contains the parseXfile function which reads the information from the DirectX (.x) file and loads each mesh-List or mesh hierarchy from within the x file into a Mesh-List. The Mesh also contains a pointer to the Frames which DirectX meshes use to store duplicate mesh information or the position, scaling and rotation of subsets within the mesh.

4.6.2.1.2 Mesh-List

The Mesh-List is a linked list of meshes which represents a single hierarchy of models from within a DirectX (.x) file Mesh. This may be a single 3d shape or a linked list of a series or shapes connected together to form a hierarchy. A Mesh-List may also contain the skinned mesh information used by Cyclone to wrap vertices around a bone hierarchy. The Mesh-List also contains the material information of each subset within the mesh hierarchy, identifying each subset and associating it with a Material instance.

4.6.2.1.3 Frame

A frame is used to store duplicate mesh information or to position, scale and rotate a subset of a mesh within world space relative to its position in model space after being transformed by its animation matrices within a single frame of a DirectX (.x) format mesh file.

4.6.2.1.4 Frame-List

A frame-list is a linked list of Mesh-lists used to represent the individual frames of an animated DirectX (.x) format mesh file.

4.6.2.1.5 Animation

The Animation class is a wrapper class for an Animation-Set which also contains a parseXFile function used to actually read the animation data from a DirectX (.x) file format mesh into its Animation-Set.

4.6.2.1.6 Animation-List

An Animation-List is a linked list of Rotation, Scale and Transformation keys which are exported from a 3d modelling package into the DirectX (.x) format mesh and can be associated with a particular frame of Animation.

4.6.2.1.7 Animation-Set

An Animation-Set is a linked list of Animation-Lists which describes a full named set of animations. A single DirectX (.x) format mesh can have multiple animation sets which can be called up and played to display different animations. A Character model for example may have a “Walk”, “Run” and “Idle” animation set.

4.6.3.1 NodeTreeMesh

The NodeTreeMesh is a child of the Object class and as such can represent any animated or static mesh file loaded from a DirectX (.x) file. The aim of the NodeTreeMesh is to recursively break down large meshes, such as those used for world geometry or terrain, into sub-groups of polygons within a specified quadrilateral or cube shape and sorting the groups into a tree structure. The NodeTreeMesh can then render only the parts (or sub-groups) of the large mesh which are contained in a given view frustum. This can dramatically reduce the number of polygons being sent to the Subset Manager (see later section) and ultimately decrease the number of polygons being handled by the game at any given time, vastly increasing rendering speed.

4.7 Physics

Cyclone provides a scalable physics implementation with use of the Tokamak Game Physics SDK. Tokamak is a free real-time physics library designed specially for games with the following features:

"**Joints**

Tokamak supports a variety of joint types and joint limits.

Friction

Tokamak provides a very realistic friction model and its computational cost is minimal.

Stacking

Tokamak is specially optimized for stacking large number of objects; one of the most frequently requested features by game developers.

Collision Detection

Build within Tokamak is a very efficient collision detection module. Currently, Tokamak provides collision detection for primitives (box, sphere, capsule), combination of primitives, and arbitrary static triangle mesh. Tokamak also supports convex to convex hull collision detection.

Rigid Particle

A lightweight rigid body providing particle effects in games at minimal cost.

Breakage

Constructing model which will break when collision occurs is a breeze in Tokamak. Fragment of original model will automatically be spawn by Tokamak's built-in breakage functionality."

(David Lam (2005) Tokamak Physics Introduction) The Tokomak SDK is a Library of functions independent of any graphics API or rendering system.

The integration of the Physics SDK with Cyclone begins with the Physics Manager which stored the actual physics simulator and classes to handle physics objects.

4.7.1 Physics Manager

The physics manager is singleton class which sits alongside the mesh, texture, sound, material and subset manager instances in the engines Base instance. The purpose of the physics manager is to hold the actual Tokomak physics simulation object. The Physics manager also contains two vectors of pointers to the rigid and animated body objects within the simulation. This allows the actual physics object within the body to be updated even when the model is out of the view frustum. This is important because an object controlled by physics may leave and enter the view frustum at any time. When the physics object is outside the view frustum, it is updated but nothing is rendered.

4.7.2 Physics Object

The abstract physics object class is a child of the engines main Object class and contains the common methods and attributes of the animated and rigid body objects. As a child of the engines main Object class can load meshes, calculates their bounding box, bounding sphere and bounding cylinder. The Physics object contains an neGeometry object which is used as a bounding volume for the mesh within the physics simulation. The Physics Object also overrides the Objects drawMeshList function to set the physics objects world matrix to that of the neGeometry object.

4.7.3 RigidBody Object

The rigid body object is a child of the physics object but contains a Tokomak neRigidBody object. The rigid body's world matrix is updated when the physics manger is updated. That matrix is then applied to the mesh before it is rendered in the drawMeshList function. A rigid body is a solid physical object with a mass which is affected by forces such as gravity. In this implementation it is represented by either the bounding box, sphere or cylinder calculated in the parent physics object.

4.7.4 AnimatedBody Object.

The animated body object is a child of the physics object but contains a Tokomak neAnimatedBody object. The animated body's world matrix is updated when the physics manger is updated. That matrix is then applied to the mesh before it is rendered in the drawMeshList function. An animated body is a solid physical object that is not affected by gravity or other forces but can be positioned within the scene.

4.7.5 Combining Physics with the scene

In order to integrate the physics objects into the Scene, two additional parameters were added to the scene file to save the mass of the object and its bounding type (1 being a box, 2 a sphere and 3 a cylinder). If the object is an AnimatedBody, it will use the bounding type to create its bounds when the scene is loaded. If the object is a Rigidbody, it will load its bounding type and its Mass from the scene file. This allows physics objects to be added to the scene without editing any code.

Before Tokomak can create a physics simulator, it must be told how many animated and rigid body objects will be in the simulation. For this reason, the Scene does a preliminary scan of the scene file. When any animated or rigid bodies are found in the file, it counts them. The totals are then sent to the physics manager to initialise it. If there are no physics objects found, the physics manager is not initialised. This would save a huge amount of memory for any game developed without physics.

The Physics simulation is a useful addition to the functionality of the Cyclone game engine. Complex behaviour can now be expressed by simple objects with relatively little effort. This adds scope to the possible playability of games developed in the engine. Tokomak also has many more promising features which are yet to be integrated such as constraints (hinges and ball-sockets for joints,

ropes, doors and other objects), physics based particles and “breakable” geometry, all of which could be beneficial if integrated in the future.

4.8 Sound

Cyclone provides a common interface to sound on Xbox and PC. A 3d sound is a sound which has a listener and emitter position. The volume of the sound is then calculated by the distance between the emitter and the receiver. To play 2d sounds, the emitter is placed in the same position as the receiver. The Sound class encapsulates the loading an playback of sounds from the Wav, Midi and wma file formats.

4.9 Input

The input in Cyclone is the one area where the Xbox and PC differ substantially. The Xbox can accept input from four control pads whereas the PC can accept input from a keyboard and mouse or a joystick. Also, the XDK version of DirectX does not contain the DirectInput library used by the PC implementation but replaces it with a custom XInput library. As with the Text, the solution to this was to provide two separate Input classes with a common interface where the same method calls will result in reading the Input from the Xbox controllers or the PC devices. However the PC input is not restricted to the Xbox controller access methods. A single instance of the input class (for the correct platform) is created and stored within the engines Base instance. The correct input class is chosen at compile time by pre-processor code within a shared “Input.h”.

4.9.1 Xbox Input

The Xbox Input class stores an array of four XBGAMEPAD structures, which are derived from the XInput Gamepad class. The XInput Gamepad class contains all of the arrays used to store the state of the various buttons on a control pad plus the methods needed to read the input from the hardware. The Cyclone XBGAMEPAD inherits that behaviour and adds an instance of an XInput rumble and feedback which allow the speeds of the control pads rumble motors to be

set, an XInput Capabilities instance which describes the capabilities of the device in the port (in this case only control pads are supported), a handle to the actual input device, and two Boolean flags to say if the pad has just been inserted or has been removed.

```
//Struct to hold custom pad data
struct XBGAMEPAD : public XINPUT_GAMEPAD
{
    // Rumble properties
    XINPUT_RUMBLE   Rumble;
    XINPUT_FEEDBACK Feedback;

    // Device properties
    XINPUT_CAPABILITIES caps;
    HANDLE      hDevice;

    // Flags for whether game pad was just inserted or removed
    BOOL        bInserted;
    BOOL        bRemoved;
};
```

Each frame the four XBGAMEPAD structures are polled by calling the getInput() method of base->input. The state of each XBGAMEPAD can then be tested from outside the Input class via the public access methods.

Xbox Controller

The Xbox controller has two types of button, Analogue and digital. The left and right thumb sticks and the six main buttons (A, B, X, Y, Black and White) on the face of the controller are analogue. As are the L and R triggers on the back of the controller. The Directional D-pad (below the left analogue stick) and the start and back buttons (to the left of the d-pad) are digital. Analogue buttons return a value ranging from 0 (off) to 255 completely on. Digital buttons are represented by true (on) and false (off).

The Xbox Input class can also be used to output force-feedback via the control pads rumble motors. The rumble motors in the left and right side of the control pad work by spinning a weight at varying speeds for varying durations to crate a vibration. This is used to give feedback within a game, such as the kickback from a gun or the vibration of a cars engine.

```
//Feedback
void Rumble(int padNumber = 0,                    // Makes the specified pad rumble with a specified
    float LeftMotorSpeedPercent = 50,             // percentage of power for each of the two spinning
    float RightMotorSpeedPercent = 50);           // motors in the gamepad.
void stopRumbling(int padNumber = 0);             // Stops the specified gamepad's Rumble motors spinning
                                                  //(passes a 0 value to both motors)
```

4.9.2 PC Input

The PC input class uses a DirectInput device to read the state of any input devices connected to the computer. The Class establishes which devices are connected when it is initialised during the initialisation of the Base class. All connected devices are polled every frame by calling the getInput() method.

4.9.2.1 Keyboard

If a keyboard is connected to the computer, the input class binds it an LPDIRECTINPUTDEVICE8 device pointer to the main DirectInput device which can then read the input from the keyboard driver. The state of each key is stored in an array of 255 characters, with one byte representing each key on the keyboard.

4.9.2.1 Mouse

Similar to the keyboard, the Input class binds its main DirectInput device to the mouse driver via an LPDIRECTINPUTDEVICE8 input device pointer. When the device identifies itself as a mouse, its input can be read into the mouse input device handler which is a default DirectX DIMOUSESTATE instance. The DIMOUSESTATE basically stores a byte for the difference in (delta) X,Y and Z of the mouse, with z being the mouse wheel, since the last frame. The mouse state also contains a set of Booleans to say which mouse buttons are down. While the

delta of the mouse is useful in a full-screen application, in a Windowed game it can become awkward. The mouse therefore uses a pointer to the Bases window handle to see if the application is windowed or full-screen. If the application is windowed then the window handle is used to get the X and Y of the mouse within the window-space which is much more useful.

4.9.2.1 Joystick

The PC Input can also handle a joystick. This is similar to the mouse and keyboard but the joystick device is bound to a DIJOYSTATE instance. If there is no joystick connected to the computer none is bound and it is not read.

4.9.3 Common Input Interface

In order to make it possible to get the input on PC and Xbox with exactly the same method calls, a common input interface was created. The common interface is more closely associated with the Xbox input then the PC. Basically each input access method from the Xbox is mirrored in the PC input interface. For example because the Xbox input class has the method:

```
float AButton(int padNumber = 0);
```

Which returns if the A button on the specified pad is down. The PC input therefore must provide an alternative for this method with the same signature to avoid any discrepancies between the two input classes. The input classes also have the same constructor and destructor signature so the Base class creates and destroys the correct input without knowing that there is a difference between the two Classes.

5.0 Resource Managers

The main aims of the resource manager layer of the Engine is to save as much memory as possible and to provide a single point of reference for each type of resource assessable to every class with a Base reference. The resource

managers are also responsible for the memory management of their specific resource type and storing any interfaces which should be accessed by all members of their managed resource type. All managers are derived from the Singleton template and stored in the Base and created and destroyed with the Base instance.

5.1. Sprite Manager

The aim of the sprite manager is to make sure that there are no repeat instances of sprites stored in memory unless the programmer specifically wants to use repeats. The sprite manager also contains the single sprite interface (an ID3DXSprite) used to blit all sprites on-screen in a single pass. Like the other managers, the sprite manager contains a private vector of sprite pointers which represents its contents. Sprites may be added to the contents of the sprite manager either as a new Instance or a Copy. Adding an instance will search through the contents of the sprite manager and if it finds a sprite with matches the sprite being added, will return a pointer to the one which already exists. This means that only one sprite is actually stored in memory but multiple references to it are stored within the sprite managers' contents. Alternatively a new Copy can be added to the sprite manager whereby a new sprite instance is pushed back regardless of whether a matching sprite already exists.

5.2. Sound Manager

The sound manager is very similar to the sprite manager except it stores sounds instead of sprites. The sound manager also stores the sound interface common to all sound instances (as the sprite manager stored the sprite interface), which consists of an ID3DXSOUNDLOADER and an ID3DXSOUNDPERFORMANCE. Each Sound instance takes a pointer to the Loader and Performance stored in the sound manager so only one instance of them need exist. As with the sprite manager a sound may be added either as a Copy or an Instance.

5.3. Physics Manager

The Physics manager was described briefly in the earlier section on Physics. The Physics manager is very different to the sprite and sound managers in that while it does contain vectors of contents and a single shared interface, it actually stores two types of object and exhibits different behaviours to the other managers. The Physics mangers main purpose is to contain the single neSimulator Tokomaks Physics simulation object as well as pointers to any AnimatedBody or RigidBody objects. The Physics manager is also responsible for creating the initial simulation and updating any objects that it contains within the simulation. Unlike the other managers, the Tokamak simulation is unable to add new physics objects at runtime as all objects must be specified when the simulation is created. Subsequently any Animated or Rigid body automatically checks that the simulation in the physics manager exists but has not began before it can construct and add a pointer to itself to the physics manager. If the simulation does not exist or has already started, then the program displays a message box.

5.4. Animation Manager

The Animation manger is a more conventional resource manger which stores the Animations defined in the Object.h file. An Animation is identified both by an Animation name and the filename of the mesh it was loaded from. As with most of the conventional managers, a vector of Animation pointers is stored within the manager, and the programmer can chose to add an instance or a copy of any given animation to the manager.

5.5. Mesh Manager

The Mesh manager is also a conventional manager in that it stores a vector of Mesh pointers (as defined in Object.h) identified by the filename from which they were loaded. Like with the Animation manager, the Meshes in the Mesh Manger are loaded automatically when Objects are created and destroyed when the Mesh Manger is destroyed at the end of runtime. The mesh manager like all other managers also logs when it destroys items using the Base logInfo method

which writes information to the Log.html file stored in the projects working directory. As with most managers Meshes can be added as Instances or Copies.

5.6. Texture Manager

The Texture manager is similar to the standard managers in that it stores a vector of textures which can be added to via the addCopy or addInstance methods. The textures loaded into the texture manager can either come from those loaded into Meshes from DirectX (.x) format files, from textures specified within Effects (see later section), from those specified in a material library (see earlier section) or even textures loaded in directly from a games source code.

5.6.1 Textures

The texture class is a small class containing a TEXTURE instance which is a platform independent pointer to a DirectX texture instance defined in Cyclone.h. A texture is a 2d bitmap image which can be wrapped around a 3d object to give surface detail to the model. The texture also contains a string used to store the filename from which the texture was loaded; this is used as a unique identifier for each texture.

5.7 Cube Map Manager

The CubeMap manager is exactly the same as the Texture manager except for the fact that it stores pointers to Cube Maps rather then Textures. As with the standard managers, the Cube Map manager can be added to via the addCopy or addInstance methods. Unlike the Texture manager, the Cube Maps loaded into the Cube Map manager can only come from Cube Maps specified within Effects (see later section) or those loaded in directly from a games source code.

5.7.1 Cube Maps

A Cube Map texture is a special texture in which six textures (mipmaps) of equal sizes are stored within a single bitmap file to represent the six faces of cube. Cube Maps are commonly used in simplified reflection models or for skyboxes. In Cyclone Cube Maps are loaded using the D3DXCreateCubeTextureFromFile

function and stored within the Cube Map class. The Cube Map class contains a platform independent pointer to a Cube Map texture (CUBEMAPTEXTURE as defined in Cyclone.h) and a string containing the filename from which the Cube Map was loaded which is used as a unique identifier.

5.8 Volume Texture Manager

Again the Volume Texture manager is exactly the same as the Cube Map manager except for the fact that it stores pointers to Volume Textures rather then Cube Maps. As with the standard managers, the Volume Texture manager can be added to via the addCopy or addInstance methods. Volume Textures can only come from those specified within Effects (see later section) or those loaded in directly from a games source code.

5.8.1 Volume Textures

A volume texture is a series of equally sized mipmaps stored within a single bitmap which represent a series of slices through a solid volume – similar to those created by a radiological cat scanner. Due to their complexity, Volume Textures are only supported in DirectX 9 so the Volume Texture class contains a DirectX 9 only pointer to a VOLUMETEXTURE instance (as specified in Cyclone.h). As they are unsupported on the Xbox, volume textures are ignored with pre-processor code.

5.9. Effect Manager

The Effect manager is similar to the other standard managers except it stores pointers to instances of the Effect class. Effects can be specified within program source code or they can be loaded in from a material library file (see earlier section).

5.9.1 Effects

The Effect class is one of the major areas of development in Cyclone. The Effect class is responsible for loading, compiling, storing and updating a single effect from a DirectX Effect (.fx) file.

5.9.2 FX Files

An Effect File (.fx) is a text document which can contain:

- global variables which can be set in the effect or in the application.
- Texture stage and sampler states.
- Multiple rendering options known as techniques – each technique can encapsulate one or more passes and the functions required to run a shader on their target hardware.
- Device states which can be set and restored before and after passes.

5.9.3 DirectX Standard Annotation and Semantics

“When Microsoft created Direc3DX Effects, it was designed to be as open and versatile as possible. To drive a more general operating logic and data input mechanism, Microsoft later added a DirectX Standard Annotation and Semantics (DXSAS) Specification. DXSAS offers three major components: semantics that categorize the content of a variable more then the already available shader input and output semantics, and scripts.” (Wolfgane Engel (2004), Programming Pixel and Vertex Shaders)

5.9.3.1 Semantics

With the new set of semantics parameters can be labelled by an agreed word which can be understood by multiple programs. By comparing the semantic to each of the agreed words and applying the agreed value from the program effects can be treating in the same way.

For example:

Float 4x4 worldMatrix : WORLDVIEWPROJECTION;

Cyclone would read this semantic in the Initialise function of the Effect class and attribute it to the a matrix concatenated from the models world, the and the cameras view and projection matrix during the Update() method of the Effect class. DXSAS specifies most of the useful matrices and also some lighting and

colouring definitions. However it is possible to add program specific annotations and this was done in Cyclone with the addition of the CUBEMAP, TEXTURE, and VOLUMETEXTURE semantics which, along with an annotation tell Cyclone which type of texture to load.

5.9.3.2 Annotations:

“Annotations are not used by the DirectX framework but can be used by the application. An example of an annotation might be to hint to associate a file with a texture type, which is defined in a string...

<string filename = “Colormap.dds”;>”

(Wolfgane Engel (2004), Programming Pixel and Vertex Shaders)

This annotation combined with the TEXTURE semantic on the declaration of a texture would tell Cyclone to load a texture into the texture manager and pass it back into the constant of the effect:

```
texture SkyColour_Tex : TEXTURE
<
        string ResourceName = "Shaders\\Skybox\\SkyColours.dds";
>;
```

In the above example, the texture in the file “Shaders\\Skybox\\SkyColours.dds” would be loaded. It should be noticed that Cyclone uses the base->getMedia() function to load media specified in shaders from a relative media path rather then forcing an absolute path.

5.9.7 Loading Effects

The aim of the lading function in the Effect class is to attempt to compile a given effect file into an ID3DXEffect and to store its semantics in the customised semantics vector which can be linked to program variables once the effect is compiled. The load function must also fill in any variables from with the effect which do not change on a per-frame basis.

5.9.8 Updating Effects

If the effect compiles and the subset manager gets into the effect part of its rendering loop, the update method is called. The parameter of the update method is the current subsets world matrix. This is used alongside the base camera's View and projection matrix to calculate and apply any matrix required within the effect. The update method can also pass the current time into an effect which uses the SAS "Time0_X" semantic and the cameras position if the effect uses the "VIEWPOSITION" semantic.

6.0. Material Manager

The Material Manager is a standard resource manager that contains pointers to instances of the Material Class.

6.0.1 Materials

```
class Material
{
    public:

        Material();
        ~Material();

        bool setEffect(LPCSTR effectFile);
        Effect* getEffect();

        bool setTexture(LPCSTR textureFile);
        Texture* getTexture();

        MATERIAL* material;
        Texture* texture;
        Effect* effect;


        Base* base;

};
```

A material is used to describe the surface of a subset (see next section). Each subsets material can be described with the minimum of a single vertex colour, which is either imported from the DirectX (.x) format mesh or defined in a material library entry. The vertex colour is stored within a MATERIAL instance

which is a pointer to a platform independent DirectX material structure. A Material may also have a texture associated with it which is applied on top of the vertex material within the fixed function pipeline. The Texture can also be loaded from the Mesh file or from a material library entry. Finally a material can have an optional effect instance associated with it, this will describe how the vertices and pixels of the mesh are rendered on the by the programmable pipeline.

While a material can describe a very simple or very complex subset surface, the final representation of the surface is decided by the Subset Manager to suit the specific hardware as best as possible. Also because the materials are stored within the Material Manager and only referenced within each subset any given loaded mesh, it is possible to replace materials in real-time by simply switching the reference within the mesh subset to another material in the Material Manager.

6.1. Subset Manager

The subset manager is the heart of Cyclones rendering capabilities. It is responsible for drawing all 3d objects seen on screen. The aim of the Subset manager is to draw any visible subsets to the screen in the most efficient way possible for the specific hardware on which the engine is running.

6.1.1 Subsets

```
class Subset
{

    public:

        Subset(ID3DXMesh* mes = NULL, Material* mater = NULL, int sub = NULL,
            D3DXMATRIX* wrld = NULL);

        ~Subset();

        ID3DXMesh* mesh;
        Material* material;
        int SubsetNumber;
        D3DXMATRIX worldMatrix;

};
```

A Subset is a set of polygons within a particular mesh which share the same material. Along with a pointer to the mesh which the subsets are associated with, a pointer to the material which is to be applied the index of the subset; each subset must also contain its final world matrix. The world matrix must be supplied so that it can either be set for each subset before it is rendered on the fixed function pipeline, or passed into the effect of subsets material if one is presnet.

6.1.2 Rendering

When an Object is told to render it pushes each of its subsets into the subset manager where they are accumulated in a vector of Subsets ready to be sorted and eventually drawn to the screen. When a subset is pushed into the subset manager, its Material's vertex colours power is set to 1 indicating that it is a visible material. The rendering loop groups all objects of the same material together by looping through all visible materials and drawing all subsets which have that material at the same time in a single batch. If the material has no effect, the D3DDevice texture and materials are set to those described in the current material and all subsets with the current material are drawn. However if the material has an Effect, the Effect is updated and the Manager loops through each pass of the Effect drawing the subset. This is a highly optimised rendering strategy as one material change is said to take the equivalent computing power of drawing 1000 polygons.

The subset manger also has a fallback mechanism. If no techniques in the current effect evaluate for the current hardware, the effect is flagged as invalid, the user is informed via a message box and the subset manager drops back to standard transformation and lighting for the invalid material in the next frame.

Once the subset manager has gone through all of the materials and the 3d world has been rendered, it can either be cleared or rendered again depending on how the programmer wants to use it. Keeping the current subsets in the manager can be used for differed shading techniques such as motion blur or HDR blooming.

7. Testing

So far the Cyclone Game Engine has been used in two separate game projects, producing two completely different games.

7.1 Town of the Dead

Town of the Dead is the name of a team project which was undertaken by a group of seven game programming students, six of whom had had no previous experience using the Cyclone Engine or programming for the Xbox. With very little assistance, the team was able to create a complex 3d survival horror game in a relatively short amount of time without access to an Xbox development kit. When it came to the end of the project, the team were surprised to see that their work ran perfectly on the Xbox without any need to port or change any code.

A copy of Town of the Dead (running on a slightly earlier version of Cyclone) is contained on the attached CD in the “D:\Cyclone\Town of the Dead\” directory. More information on Town of the Dead can be found in the games brief use manual in **Appendix 3** or on the games website at :
http://www.acidgames.net/TownOfTheDead

7.2 Viewtopia

Viewtopia is the name of the program used to test and display the Cyclone Game Engine in the presentation for this project. The aim of Viewtopia was to demonstrate some of the core aspects of the engine by creating a real-time material editing application within a first-person 3d environment. Viewtopia is contained on the attached CD in the “D:\Cyclone\Viewtopia\” directory. An instruction manual for Viewtopia is also included in **Appendix 4**.

By creating an interface for the Mesh and Material mangers and using render targets, Viewtopia also demonstrated how parts of the engine could be extended upon.

8. Conclusions, Evaluation & Further Work

The main problem which Cyclone addressed was in providing a stable cross-platform solution which would allow none-Xbox literate programmers to develop for the Xbox without the need for access to an expensive development kit or any need to “port” code. This aspect of the engine was proven to be successful when the team of developers created the Town of the Dead game.

Another main feature of the engine was the material, effect and subset management system. I feel that this despite its initial complexity, this aspect of the engine has become very stable and very efficient. While I am pleased with the PC support for HLSL effects, I feel that with more time it would be possible to build Xbox support into the effect system. The main problem with adding Xbox support is that the Xbox only supports its own XPS and XVS assembly language effects which must be compiled with an external compiler prior to runtime. While it would be nice to include shaders on the Xbox, it is not essential and I think the engine does a good job of finding the “best case” solution.

One of the other areas I am happy with is the physics library. Physics were not specified in the initial design of Cyclone but have provided a worthwhile and interesting addition to the engine. I am very much interested in extending the physics support to offer more complex simulations including breakable bodies, and constraints.

The development of Cyclone has been highly successful as all of the criteria set at the beginning of the project have been met and even extra features have been added. Cyclone was a highly ambitious project which took over nine months of solid work to complete. I am pleased with the overall product as it stands up very well with some industry standard engines and even offers some innovative material editing features, more often found in 3d modelling software.

9. References

Steven Goodwin (2005): **Cross-Platform Game Programming**:
Chapter 1 page 22
ISBN: 01-58450-379-3.

Joseph Meyers (2006): **Game Engine Definition**
[Online] <http://en.wikipedia.org/wiki/Game_engine >
[Accessed: April 2006].

“HellChick” (1/5/2001): **What’s a Game Engine?**
[Online] <http://www.3dactionplanet.com/features/articles/gameengines/index.shtml >
[Accessed: April 2006].

Pixar (2006): **RenderMan - Artist Tools - What's Renderman**
[online]<https://renderman.pixar.com>
[Accessed April 2006].

Steve Steering (11/2005): **Ogre Manual**
[Online] < http://www.ogre3d.org/docs/manual >
[Accessed: April 2006].

Wikkipedia (2006): **Game Engine Definition**
[Online] <http://en.wikipedia.org/wiki/Game_engine >
[Accessed: April 2006].

Wolfgang Engel (2004): **Programming Pixel and Vertex Shaders**:
Chapter 1 page 79
ISBN: 1-58450-349-1

IGN Staff (2/10/2001): **Xbox Specification**
[Online] <http://gear.ign.com/articles/306/306618p1.html>
[Accessed: April 2006].

Wikkipedia (2004): **Viewing Frustum Definition**
[Online] < http://en.wikipedia.org/wiki/Viewing_frustum>
[Accessed: April 2006].

David Lam (21/03/2005): **Tokamak Physics Introduction**
[Online] <http://www.tokamakphysics.com/intro.htm>
[Accessed: April 2006].

MSDN (2005): **Matrix and Vector Explanations**
[Online] < msdn.microsoft.com >
[Accessed: April 2006].

Referenced Images:

http://www.zfx.info/Data/Tutorials/Zerbst/shader/vertex_alu.gif
http://www.zfx.info/Data/Tutorials/Zerbst/shader/pixel_alu.gif
http://www.ogre3d.org/docs/manual/images/uml-overview.png
http://www.lighthouse3d.com/opengl/viewfrustum/images/vf.gif

10. Bibliography

Cross-Platform Game Programming

Steven Goodwin

2005: Charles River Media, Inc.

ISBN: 1-58450-379-3

Game Programming Gems

Mark DeLoura

2000: Charles River Media, Inc.

ISBN: 1-58450-049-2

Game Programming Gems 2

Mark DeLoura

2001: Charles River Media, Inc.

ISBN: 1-58450-054-9

Game Programming Gems 3

Mark DeLoura

2002: Charles River Media, Inc.

ISBN: 1-58450-233-9

Game Programming Gems 4

Mark DeLoura

2004: Charles River Media, Inc.

ISBN: 1-58450-295-9

Game Programming Gems 5

Mark DeLoura

2005: Charles River Media, Inc.

ISBN: 1-58450-352-1

Programming Vertex and Pixel Shaders

Wolfgang Engel

2004: Charles River Media, Inc.

ISBN: 1-58450-349-1

ShaderX^3 Advanced Rendering with DirectX and OpenGL

Wolfgang Engel

2005: Charles River Media, Inc.

ISBN: 1-58450-357-2

Mathematics for 3D Game Programming & Computer Graphics

Eric Lengyel

2004: Charles River Media, Inc.

ISBN: 1-58450-277-0

Real-Time 3d Terrain Engines using C++ and DirectX 9

Greg Snook

2003: Charles River Media, Inc.

ISBN: 1-58450-204-5

3D Game Engine Design

David H. Eberly

2003: Morgan Kaufmann Publishers.

ISBN: 1-55860-593-2

Game Physics

David H. Eberly

2004: Morgan Kaufmann Publishers.

ISBN: 1-55860-740-4

11. Appendices

Appendix 1: Xbox Hardware Specification

Central Processing Unit (CPU): 733 MHz chip created by Intel
Graphics Processing Unit (GPU): 250MHz custom chip named XGPU, developed by Microsoft and nVIDIA
Total Memory: The RAM in the Xbox is 64 MB running at 200MHz DDR (Double-Data-Rate) supplied by Micron
Memory Bandwidth: 6.4 GB/sec
Polygon Performance: 125 M/sec
Sustained Polygon Performance: 100+ M/sec (transformed and lit polygons per second)
Micropolygons/particles per second: 125 M/sec
Particle Performance: 125 M/sec
Simultaneous Textures: 4
Pixel Fill Rate - No Texture: 4.0G/Sec (anti-aliased)
Pixel Fill Rate - 1 Texture: 4.0 G/Sec (anti-aliased)
Compressed Textures: Yes (6:1)
Full Scene Anti-Alias: Yes
Micro Polygon Support: Yes
Storage Medium: 2-5x DVD, 10GB hard disk, 8MB memory card
I/0: 2-5x DVD, 10GB hard disk, 8MB memory card
Audio Channels: 64 (up to 256 stereo voices)
3D Audio Support: Yes
MIDI DLS2 Support: Yes
AC3 Encoded Game Audio: Yes
Broadband Enabled: Yes
Modem Enabled: No
DVD Movie Playback: Remote control package required
Maximum Resolution: 1920x1080
Maximum Resolution: (2x32bpp frame buffers +Z): 1920x1080
HDTV Support: Yes
Controller Ports: 4 USB Ports

(IGN Staff(2001), Xbox Specifications)

Appendix 3: Town of the Dead User Manual

Town of the Dead is a survival horror game set in the university of Huddersfield. The objective of the game is to escape the zombie infested Canalside West building by solving puzzles and fighting off the zombie hordes.

Controlling the player:

The player is moved using the arrow keys. Right and left rotate the player while up and down move the player forwards or backwards. If the player has a gun equipped, he can shoot zombies by using the control key to lock on and shift key to fire. Pressing the SPACE key over an item or a door will activate its dialog.

Dialogs

Using the arrow keys highlights an option and the space key activates it.

Inventory

Pressing the i key or picking up an item brings the player to the inventory. In the inventory, the player can arrange items by combining similar items and swapping items around to maximise the available space. Pressing SPACE twice on an item also brigs up the menu to drop and equip items.

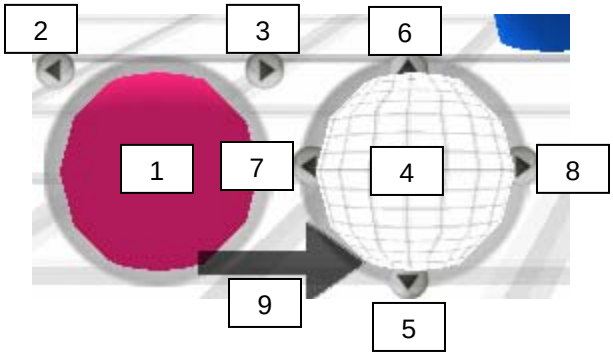
Map

The map of the Canalside West building can be displayed by pressing the M key at any time during the game. The map shows where the player is and the state of any doors or items which have been checked.

Appendix 4 : Viewtopia User Manual

This section describes the basic controls used in the Viewtopia demonstration included on the CD.

Using the Menus:



1: Selected Material Preview.

2 - 3: Clicked to scroll through the materials in the material manager.

4: Selected Subset Preview.

5 - 6: Clicked to scroll through Meshes in the Mesh Manger.

7 - 8: Clicked to scroll through the subsets of the selected mesh.

9: Clicked to apply the material on the left to the subset previewed on the right.

Controlling the camera:

The camera can be controlled using the **arrow keys** and rotated using the **Home**, **End**, **Delete** and **Page Down** keys.

Appendix 5: Code Listing

Cyclones code is fully documented and available on the attached CD. A code comment report is also on the CD under D:\Documentation\Comment Report\ or http://www.acidgames.net/Cyclone/Source.